JavaGenes: Evolving Graphs with Crossover

Al Globus, Veridian-MRJ Technology Solutions, Inc. at NASA Ames Research Center
Sean Atsatt, Sierra Imaging, Inc.
John Lawton, University of California at Santa Cruz
Todd Wipke, University of California at Santa Cruz

Abstract

Genetic algorithms usually use string or tree representations. We have developed a novel crossover operator for a directed and undirected graph representation, and used this operator to evolve molecules and circuits. Unlike strings or trees, a single point in the representation cannot divide every possible graph into two parts, because graphs may contain cycles. Thus, the crossover operator is non-trivial. A steady-state, tournament selection genetic algorithm code (JavaGenes) was written to implement and test the graph crossover operator. All runs were executed by cycle-scavaging on networked workstations using the Condor batch processing system. The JavaGenes code has evolved pharmaceutical drug molecules and simple digital circuits. Results to date suggest that JavaGenes can evolve moderate sized drug molecules and very small circuits in reasonable time. The algorithm has greater difficulty with somewhat larger circuits, suggesting that directed graphs (circuits) are more difficult to evolve than undirected graphs (molecules), although necessary differences in the crossover operator may also explain the results. In principle, JavaGenes should be able to evolve other graph-representable systems, such as transportation networks, metabolic pathways, and computer networks. However, large graphs evolve *significantly* slower than smaller graphs, presumably because the space-of-all-graphs explodes combinatorially with graph size. Since the representation strongly affects genetic algorithm performance, adding graphs to the evolutionary programmer's bag-of-tricks should be beneficial. Also, since graph evolution operates directly on the phenotype, the genotype-phenotype translation step, common in genetic algorithm work, is eliminated.

Introduction

[Holland 1975] applied the principles of Darwinian evolution to searching through the space of fixed length binary strings representing solutions to various problems. This technique, genetic algorithms, has been applied to a wide variety of fixed and variable lengthed strings of many kinds of symbols: for example, bits, characters, integers, real numbers, etc. [Koza 1992] extended this technique to hierarchical trees representing computer programs; this is usually called genetic programming. Both genetic algorithms and genetic programming use crossover and mutation to evolve solutions to problems. Related techniques, evolutionary algorithms and evolutionary programming, use only mutation to evolve solutions. For an excellent review of evolutionary software techniques as of the spring of 1997 see [Baeck, et al. 1997]. Evolutionary software has been applied to a wide variety of problems as evidenced by a voluminous literature and many conferences devoted to the subject.

Genetic algorithms seek to mimic natural evolution's ability to produce highly functional objects. Natural evolution produces organisms. Genetic algorithms produce sets of parameters, programs, molecular designs, and many other structures. Genetic algorithms usually solve problems by some variant of the following algorithm:

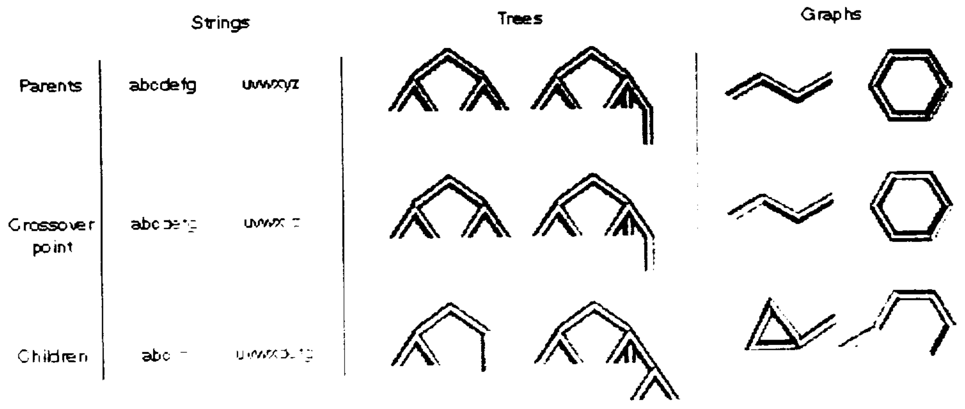
1. Randomly generate a population of individual potential solutions.
2. Evaluate each individual using a fitness function.
3. For each new generation, repeatedly select parent individuals at random with a bias towards better individuals, and create children by applying transmission operators. Transmission operators may include:
    1. Crossover: each of two parents is divided into two parts and one part from each parent is combined into a child.
    2. Mutation: a single ?parent? is randomly modified to create a child.
    3. Reproduction: a single ?parent? is copied into the new generation.
4. Continue until an acceptable solution is found or exhaustion sets in.

If genetic algorithms can be usefully applied to two data structures, strings and trees, one might ask: "Can the same techniques may be usefully extended to other data structures; for example, graphs?" In this paper, a graph is a set of vertices and a set of edges, where each edge connects two vertices. Specifically, we are *not* referring to an image representing data when we use the term "graph." Applying genetic algorithms to new data structures is important because the representation has been shown to have a strong affect on performance. As [De Jong 1990] states:

> "The key point in deciding whether or not to use genetic algorithms for a particular problem centers around the question: what is the space to be searched? If that space is well-understood and contains structure that can be exploited by special-purpose search techniques, the use of genetic algorithms is generally computationally less efficient. If the space to be searched is not so well understood, and relatively unstructured, and *if an effective GA representation of that space can be developed,* then GAs provide a surprisingly powerful search technique for large, complex spaces." (italics added by [Forrest and Mitchell 1993])

Genetic algorithms evolve populations by mutation and crossover. Graph mutation operators are fairly obvious and easy to implement. JavaGenes implements several graph mutation operators (add vertex, add edge, change vertex, change edge, etc.), but these are not the focus of this paper. Crossover is easy to implement for strings and trees because these data structures can be divided into two pieces at any point. Crossover applied to graphs is non-trivial.

Crossover on Various Data Structures



Crossover can be applied to strings, trees, and graphs. Note that only graph crossover sometimes requires breaking multiple edges, and that fragment combination must work on fragments with different numbers of broken edges.

For the purpose of our discussion, crossover may be considered to have two parts:

- Division, where each parent is divided into two fragments.
- Fragment combination, where two fragments are fused into one.

Graph crossover can be accomplished by breaking edges. However, graph crossover is complex because:

- Graph crossover cannot trivially divide the data structure at any point, because any edge may be a member of one *or more* cycles. *All* of these cycles may need to be broken to divide the graph into two pieces because the edges to break must be chosen at random to avoid biasing the search. One cannot avoid breaking edges involved in cycles, because then the cycle structure will not evolve.
- Graph fragments produced by division may have more than one crossover point ("broken edges") that requires reattachment during fragment combination.
- When two fragments are combined they may have different numbers of broken edges to be merged.
- For a graph crossover operator to potentially reach any possible graph from an initial random population, the crossover operator must be able to create and destroy individual cycles, fused cycles (cycles that share edges), cages (two or more cycles, each pair of which share at least two edges), and combinations of fused cycles and cages.

The primary contribution of this paper is to introduce a new graph crossover operator that:

- Can operate on any connected directed or undirected graph. A connected graph is one where every pair of vertices is connected by at least one set of edges.
- Divides graphs at randomly generated cut sets. A cut set is a set of edges which divides a connected graph into two parts.
- Can evolve arbitrary cyclic structures given at least some cycles in the initial population.
- Always produces connected undirected graphs.
- Almost always produces connected directed graphs.

Our graph crossover operator was applied to evolving pharmaceutical drug molecules and simple digital circuits. Molecules are represented as a set of atoms (vertices) connected by a set of bonds (edges). Digital circuits are represented as a set of devices (vertices) connected by a set of wires (edges).

Our original motivation was to evolve molecules. At first, we attempted to devise a tree representation of molecules. However, many molecules contain cycles; which chemists call rings. Therefore, any attempt to use genetic programming to design molecules must have a mechanism to evolve cycles. This is non-trivial when crossover can replace any sub-tree with some other random sub-tree. After much thought, we were unable to devise a crossover-friendly tree representation of arbitrary cyclic graphs. Crossover-friendly means that any sub-tree is a potential crossover point without restriction.

Note that the problem we are solving here involves constructing graphs, rather than examining graphs. Many classic graph theoretical problems, graph coloring, finding Hamilton circuits, graph isomorphism, and maximal subgraphs discovery, have been studied in the search literature. For example, see [Cheeseman, et al. 1991]. All of these problems investigate one or more graphs. Our problem is to find a graph representing a molecule or circuit with desirable properties. Thus, the characteristics of our search space are quite different from the classic graph problems in the literature. There is little or no reason to believe that crossover would be useful for solving these classic graph problems. It *is* possible to view the graph design problem as a search through the space of all possible graphs, and this space can be represented as an extremely large graph, as is explained in the next paragraph. However, rather than determining some property of the graph representing the search space, we simply look for one or more points in that space that possess desirable properties.

The space-of-all-graphs has very little in common with a multidimensional continuous Cartesian space. The space-of-all-graphs consists of a large number of discrete points, each of which represents a particular graph. One may define the neighbors of a graph to be all graphs that could be formed by a single mutation. These mutations might be:

- adding or deleting an edge,
- adding an edge connected to a new vertex,
- deleting a vertex and the edges connect to it,
- changing a vertex's type,
- changing an edge's type,
- or adding a vertex to the null graph. The null graph has no vertices or edges.

If each point in the space-of-all-graphs is considered to be a vertex, and each of these vertices is connected to its neighbors by edges, then the space-of-all-graphs becomes, itself, a graph. Note that this space has no derivatives. If each graph (or vertex in the space-of-all-graphs) is associate with a number, such as a fitness, then the space-of-all-graphs may have multiple, local optima if more than one graph has better fitness then all of its neighbors.

Previous work

[Weininger 1995] patented genetic algorithms for molecular design, and used the standard graph representation of a molecule in the crossover operator. The patent describes the straightforward and fairly obvious parts of mapping genetic algorithm techniques to graph-based molecular design, and the non-obvious portions: the crossover algorithm and fitness functions. The crossover algorithm described in the patent depends on two parameters: a digestion rate, which breaks bonds, and a dominance rate, which controls how many parts of each parent appear in a child. As described by figure 7 and related text in the patent, [Weininger 1995]'s crossover algorithm removes random bonds from parents according a "digestion rate" to create fragments, and does *not* connect the fragments from both parents with new bonds when forming children. A "dominance rate" determines how many fragments of each parent are placed in the child, which can obviously lead to disconnected children. *When restricted to generating connected children (covalently bound molecules), [Weininger 1995]'s crossover operator generates a child that is simply a fragment of one parent, so in this case the operation is not really crossover at all, but rather a form of mutation.* [Weininger 1995] uses the Tanimoto index (described below) as a distance measure for a number of fitness functions. Daylight Chemical Information Systems, Inc., which holds the patent, reports using genetic algorithm techniques to discover lead compounds for pharmaceutical drug development and other commercial successes.

Circuit design is another field for which genetic algorithms using a graph representation should, in principle, be well suited. Genetic algorithms using a variable length string representation [Lohn and Colombano1998] and genetic programming [Koza 1997] [Koza 1999] have been used to design analog circuits. In the genetic programming case, a tree language capable of generating a subset of the analog circuits compatible with the SPICE (Simulation Program with Integrated Circuit Emphasis) simulator [Quarles, et al. 1994] was developed. The system was used to design a lowpass filter, a crossover filter, a four-way source identification circuit, a cube root circuit, a time-optimal controller circuit, a 100 dB amplifier, a temperature-sensing circuit, and a voltage reference source circuit. Thus, genetic algorithms can design graph-structured systems. Therefore, it may be advantageous to directly evolve graphs rather than strings or trees that are be interpreted to generate cyclic graphs.

[Nachbar 1998] used genetic programming to evolve molecules for drug design by sidestepping the crossover/cycles problem and representing molecules with trees. Each tree node represented an atom with bonds to the parent-node atom and each child-node atom. Hydrogen atoms were explicitly represented and are always leaf nodes. Rings were represented by numbering certain atoms and allowing a reference to that number to be a leaf node. *Crossover was constrained not to break or form rings.* Ring evolution was enabled by specific ring opening and closing mutation operators.

[Teller 1998] reported developing a graph crossover algorithm as part of his dissertation at Carnegie Mellon University, but supplied few details. This technique was applied to Neural Programming, a system developed by Teller to combine neural nets and genetic programming.

[Globus, et al. 1999] reported results evolving pharmaceutical drug molecules with the crossover operator discussed in this paper. The crossover operator was only capable of operating on undirected graphs, not the directed graphs required for circuits. Furthermore, after publication, a bug was discovered that severely limited cycle evolution. [Globus, et al. 1999] reported adequate performance evolving small molecules but poor results evolving larger molecules with more complex cycle structures; as might be expected in hindsight. The present paper reports results evolving the same molecules with the corrected operator as well as results on undirected graphs representing digital circuits. The molecular results are significantly better. See the Results section for details.

Method

Genetic Algorithm with Graph Representation

Molecules

One approach to drug design is to find molecules similar to good drugs. Ideally, a candidate replacement drug is sufficiently similar to have the same beneficial effect, but is different enough to avoid negative side effects. To use genetic algorithms for similarity-based drug discovery, we need a good similarity measure that can score any molecule. [Carhart, et al. 1985] defined such a similarity measure, all-atom-pairs-shortest-path, and searched a large database for molecules similar to diazepam. We use this similarity measure to evolve a population of molecules towards a target molecule.

JavaGenes uses undirected graphs to represent molecules. Vertices are typed by atomic element. Edges can be single, double, or triple bonds. Valence is enforced. Heavy atoms (non-hydrogen atoms) are explicitly represented by vertices, but hydrogen atoms are implicit; i.e., any heavy atom with an unfilled valence is assumed to be bonded to hydrogen atoms, but these are not represented in the data structure.

Since we're interested in the properties the crossover operator, for this study JavaGenes evolved populations using crossover only and mutation was not used.

Each individual in the initial population was generated by the following algorithm:

```
1. atoms = random number between half and twice the number in the target molecule
2. rings = random number between half and twice the number in the target molecule
3. while (true)
    1. for some number of tries
        1. choose first atom at random
        2. for atoms-1
            1. if possible, add random atom bonded (with random bond) to a random existing
               atom, respecting valence
        3. for number of rings
            1. if possible, add random bond between two randomly chosen atoms, respecting
               valence
        4. if molecule has correct number of atoms and rings, return molecule
    2. rings--
```

The random atoms were chosen with equal probability from the elements in the target molecule. Thus, the initial population of a job searching for cholesterol would consist of roughly equal numbers of carbon and oxygen atoms. The random bonds (single, double, or triple) were chosen with equal probability from the bond types in the target molecule. The last step (rings--) is necessary because it is possible to run out of empty valence before molecular construction is complete. Consider the case where the first atom is chlorine. If the second atom is also chlorine, the two chlorine atoms must share a single bond and the valence of both atoms will be filled, so no additional atoms may be added. If the random generation algorithm fails to make a molecule with the proper number of atoms and rings, then the algorithm tries again. Some choices of "atoms" and "rings" require a molecule with more rings than is possible given the number of atoms. Consider searching for cubane. If atoms = 4 and rings = 16, it is impossible to build a molecule to the specification. Therefore, after trying to generate a molecule several times, the algorithm will reduce the number of required rings by one and try again. Since all of the target molecules contain at least one element with a valence of two or more, there is no hard limit to the number of atoms that may be in a molecule.

The number of rings, by our definition, is always equal to `bonds - atoms + 1`. For this definition, single, double, and triple bonds are counted as one bond each. This formula corresponds to the following unambiguous definition of the rings in a molecule taken from [Corey and Wipke1969]. In this definition:

1. the set of all rings must include all the bonds participating in any ring,
2. removing any ring will result in at least one bond not being included in any ring.
3. no ring may share more than half its bonds with any other ring.
4. and the set of rings chosen must be at least as large as any other set of rings with the first three properties.

While this definition is precise and useful for coding, it comes to the interesting conclusion that cubane (a cubic molecule) has five rings, not six. Consider the six sides of cubane to be the six rings in the set of rings. If any one of the six rings is removed from the set of rings, all bonds are still included in the set of rings. Therefore, only five rings are necessary to meet the definition.

Tournament selection was used to choose parents in a steady state genetic algorithm. Tournament selection means that each parent is chosen by comparing two randomly chosen individuals and taking the best. Steady state means that new individuals (children) replace poor individuals in the population rather than creating a new generation. The poor individuals are also chosen by tournament, but the worst individual is selected for replacement. By convention, after population-size individuals have been replaced, we say that one generation is complete. The implementation follows this procedure:

```
1. Generate a random population of molecules
2. Repeat many times, gathering data periodically:
    1. Select two molecules from the population at random. Call the better
       molecule father.
    2. Select two molecules from the population at random. Call the better
       molecule mother.
    3. Make a copy of father and divide it randomly into two fragments.
    4. Make a copy of mother and divide it randomly into two fragments.
    5. Combine one fragment of the copy-of-father and one fragment of the
       copy-of-mother into a molecule called son.
    6. Combine the other fragment of the copy-of-father and the other fragment of
       the copy-of-mother into a molecule called daughter.
    7. Choose two molecules from the population at random. Replace the worst one
       with son.
    8. Choose two molecules from the population at random. Replace the worst one
       with daughter.
3. Repeat until satisfied
```

Crossover requires two procedures: one to divide molecules into two fragments, and a second to combine two molecular fragments. To divide a molecule into two fragments we use the following procedure:

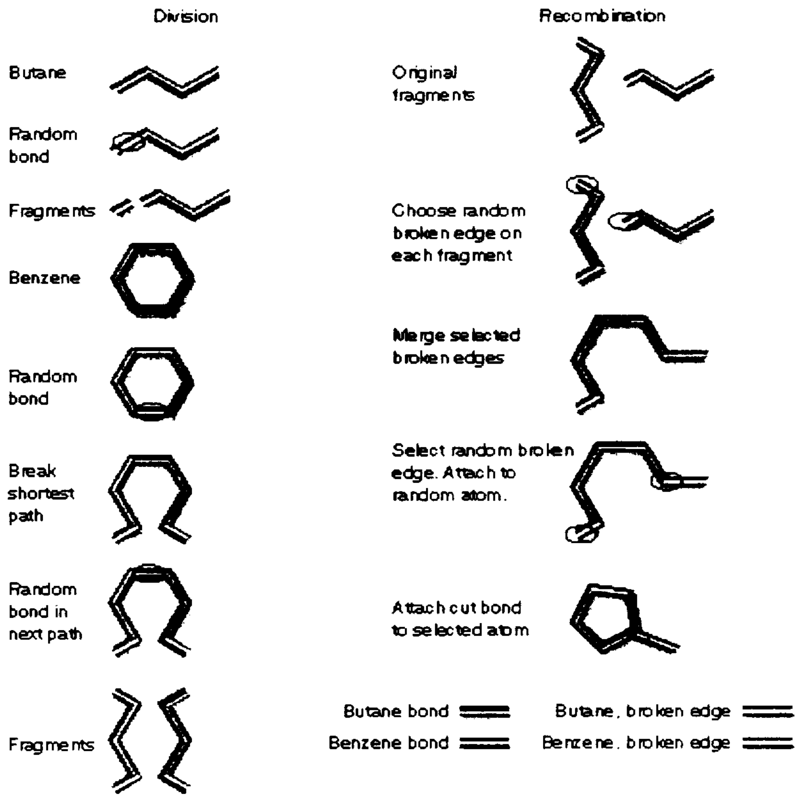
```
1. Choose an initial random bond
2. Repeat
    1. Find the shortest path between the initial bond's vertices (the first time
       this will simply be the initial bond).
    2. Remove and remember a random bond from this path. These bonds are called
       "broken edges."
3. Until a cut set is found, i.e., no path exists between the initial bond's
   vertices.
```

To combine fragments we use the following procedure:

```
1. Repeat
    1. Select a random broken edge. Determine which fragment it is associated
       with.
    2. If at least one broken edge in other fragment exists
        1. choose one at random
        2. merge the broken edges into one bond; respecting valence by reducing
           the order of the bond if necessary
    3. Else flip coin (this step was disabled by a bug in [Globus, et al. 1999])
```

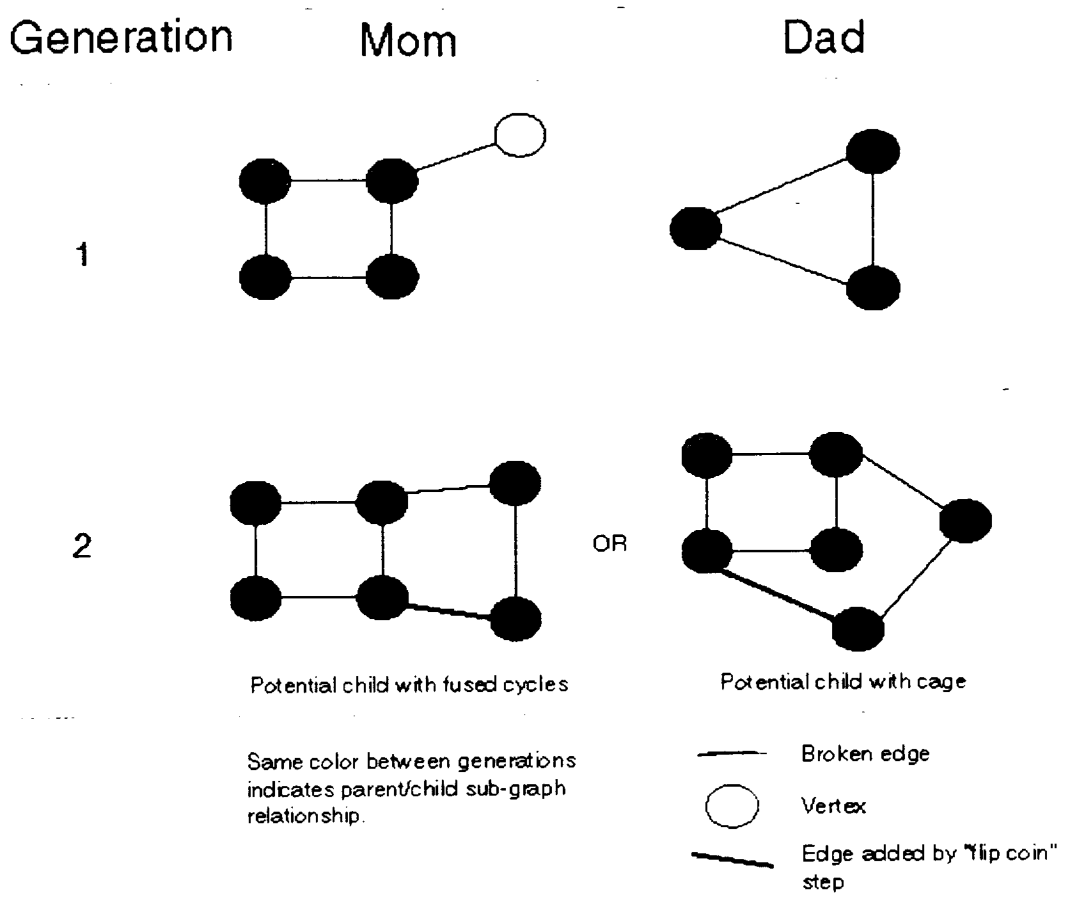
```
        1. if heads -- attach the broken edge to a random atom in other fragment
           (respecting valence)
        2. if tails -- discard the broken edge
2. Until each broken edge has been processed exactly once
```

Graph Crossover of Butane and Benzene to Create a Child



Butane and benzene are divided at random points. Then a fragment of butane and a fragment of benzene are combined. Note that benzene must be cut in two places. Also, during fragment combination the benzene fragment has more than one broken edge. A random choice is made to connect this extra broken edge to a random atom in the butane fragment. Alternatively, the extra broken edge could have been discarded.

Forming Fused Cycles and Cages with Crossover


Generation
Mom
Dad
1
2
OR
Potential child with fused cycles
Potential child with cage
Same color between generations indicates parent/child sub-graph relationship.
Broken edge
Vertex
Edge added by "flip coin" step

Graph crossover can generate fused cycles and cages. The "flip coin" step of the crossover algorithm is crucial to this functionality.

Our crossover operator can open and close rings using crossover alone, and can even generate cages and higher dimensional graph structures as long as there are rings in the population. Unfortunately, if there are no rings in a population, none can be generated. Also, once a population consists entirely of two-atom-molecules, no molecules with more than two atoms can be generated. Nonetheless, this crossover operator is the most general of those we examined or found in the literature. In particular, unlike [Nachbar 1998], no special-purpose ring opening and closing operators are necessary. Unlike [Weininger 1995], no parameters are necessary and disconnected "molecules" are never produced.

The computational resources required for genetic algorithms to find a solution is a function of the size of the search space, among other factors. The space of all possible graphs is combinatorial and enormous. For molecular design, this space can be radically reduced by enforcing valence limits for each atom. Thus, a carbon atom with one double and two single bonds will not be allowed to add another bond. Also, avoiding explicit representation of hydrogen atoms substantially reduces the size of the graph.

Digital Logic

Since our group at NASA Ames is interested in molecular electronics, and the best way to design molecular electronic circuits is unclear, we thought it might be interesting to use JavaGenes to evolve digital circuits. If reasonably successful, it might then be possible to apply JavaGenes to designing molecular circuits once simulators are available to implement appropriate fitness functions.

JavaGenes uses directed cyclic graphs to represent circuits. In other words, each vertex has a set of input edges and a set of output edges, and each edge has an input vertex and an output vertex. Each vertex and edge has a current state (usually 0 or 1). Vertices are the digital devices And, Nand, Or, Nor, Xor, and Nxor. The initial value of a device may be zero or 1. Each device can have any number (including zero) of input and output edges. If there are no input edges, And, Or, and Xor output 0, and Nand, Nor, and Nxor output 1. If there is one input edge, it is copied to output or inverted for Nand, Nor, and Nxor vertices. If there are two or more input edges:

- And will output 0 if any input value is 0, or 1 if all input values are 1.
- Or will output 0 if all input values are 0, or 1 if any input value is 1.
- Xor will output 0 if an even number of input values are 1, or 1 if an odd number of input values are 1.

For the devices whose names start with N and have two or more input edges, the output is inverted (0 becames 1, 1 becomes 0). This scheme allows vertices to have any number of input and output edges, thereby simplifying the crossover operator substantially. Nonetheless, any circuit represented this way can be easily mapped to a circuit restricted to two and three terminal devices plus fan out.

There are two special vertices in each circuit: input and output. The input vertex does no processing, but simply accepts values from a simulator and outputs those values to all output edges. No input edges are allowed. The output vertex is a digital device like the others, but has no output edges. The output vertex hands output values to a simulator.

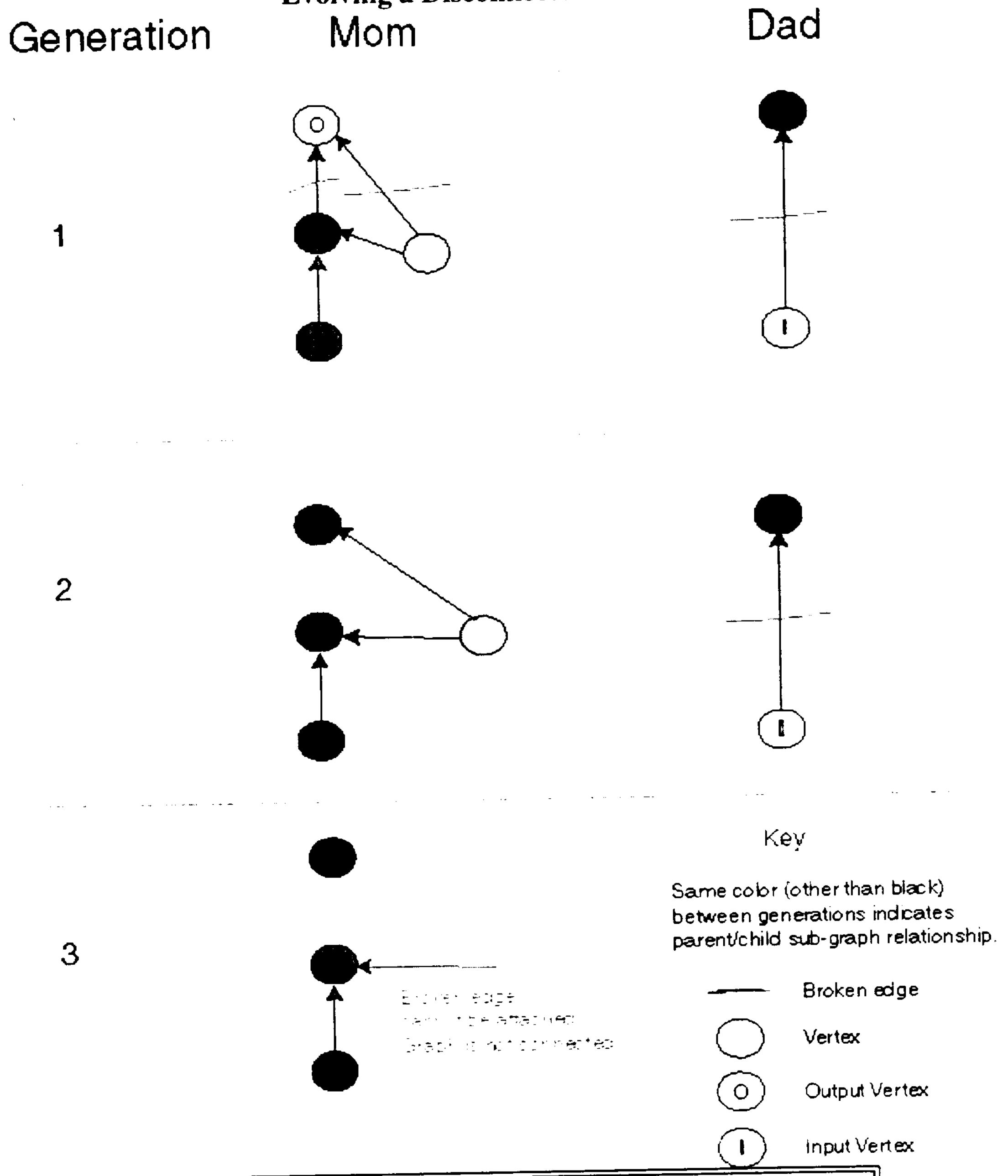
Each individual in the initial population was generated by choosing a random number of vertices and edges where the numbers fell within upper and lower bounds set by the JavaGenes input parameters. Vertex types were randomly chosen from all possible types. The circuits were created as follows:

1. Input and output vertices were created
2. The rest of the vertices were added one at a time by attaching one output edge to the input of a vertex already in the circuit.
3. An output edge from the input vertex was connected to a random vertex.
4. Edges were added at random to create the required number of cycles.

Because edges were directed and there were two special vertices in each circuit (the input and output vertices), the crossover algorithm was somewhat different than for molecules. During division, instead of choosing a random edge and cutting random edges connecting the vertices of the chosen edge, the input and output vertices were chosen and edges between *them* were broken until the graph divided into two parts. This guaranteed that each fragment had exactly one input or one output vertex, but never both. Obviously, only fragments containing an input vertex were combined with fragments containing an output vertex, and vice versa. Furthermore, during combination, broken edges with the output vertex removed were only merged with broken edges where the input vertex was removed, and vice versa.

This modified crossover operator has the interesting property that, under certain rare conditions, a disconnected circuit can be created. This anomaly requires multiple generations to occur, and is caused by the fact that the edges are directed. While we have not collected data, the anomaly appears to occur only once for every few thousand crossover operations. When this occured, we simply discarded the child.

Evolving a Disconnected Circuit


Generation
Mom
Dad
1
2
3
Key
Same color (other than black) between generations indicates parent/child sub-graph relationship.
Broken edge
Vertex
Output Vertex
Input Vertex
O
I

This sequence of evolutionary events results in a disconnected directed graph. Fortunately, this occurs rarely.

Fitness Functions

Molecules: all-pairs-shortest-path similarity

A key to successful genetic algorithm search is a good fitness function [Kinnear 1994] -- for tournament selection, a function that can determine if one molecule is better than another. This function must be very robust, since the randomly generated initial molecules rarely make much chemical sense. Fitness functions must also make fine distinctions between any two molecules, even if both are very good or very bad. These fine distinctions are necessary to avoid flat regions in the fitness space where evolution has no direction. Also, for our initial studies, we wanted a fitness function that only required the graph of a molecule, not the xyz coordinates of each atom.This simplifies initial studies and avoids the necessity of equilibrating the structure of candidate molecules, a CPU intensive step. The all-atoms-pairs-shortest-path similarity test chosen [Carhart, et al. 1985] is a robust graph-only fitness function. The fitness function algorithm was as follows:

1. Each atom is given an extended type consisting of a tuple containing the element and the number of single, double, and triple bonds the atom participates in. For example, the carbon in carbon dioxide is represented by the tuple (C,0,2,0). The C indicates that the atom is carbon. The zeros indicate that the atom participates in no single or triple bonds. The 2 indicates that the atom participates in two double bonds.
2. The shortest path between each pair of atoms is found.
3. A bag is constructed with one element for each atom pair. A bag is a set that contains duplicate elements. Each element in the bag is a tuple consisting of the sorted extended types of the two atoms and the length of the shortest path between them. For example, the path between a carbon and one oxygen in carbon dioxide would be represented by: ((C,0,2,0),(O,0,1,0),1). The first two elements in the tuple are the extended types of the atoms. The 1 is the length of the path between them.
4. The fitness of each candidate molecule is the distance between its bag and the similarly constructed bag of a target molecule. The distance measure used is the Tanimoto index. This is:

   $$|c \text{ intersection } t| \,/\, |c \text{ union } t|$$

   where c is the candidate?s bag and t is the target?s bag. Two elements are considered identical for the purpose of the intersection and union operators if the atoms have the same extended types and the distance between them is identical. Each duplicate in the bag is considered a separate element for the purpose of the intersection and union operators. The Tanimoto index always returns a number between 0 and 1. We prefer fitness functions that return lower numbers for fitter individuals, so we subtract the Tanimoto index from one to get the fitness.

Search spaces with many local optima are difficult to examine because many algorithms tend to converge to local optima and miss the global optimum. The all-pairs-shortest-path fitness function has many local optima over the space-of-all-molecules whenever the target molecule contains rings. Consider benzene, a six membered ring, as the target molecule. Any ring other than a six membered ring will be at a local optima because a bond must be removed, lowering the fitness in most cases. before bonds and atoms can be added to generate benzene. Thus, any target molecule containing rings will have an associated search space containing many local optima. Interestingly, a small modification in the definition of the space-of-all-molecules eliminates these local optima. Consider mutations that

1. replace an edge with a vertex and two edges.
2. or replace a vertex and two edges with one edge.

These mutations can change the size of rings. If these mutations are allowed to create neighbors in the space-of-all-molecules. then incorrectly sized rings are not local optima. This illustrates the difficulty of understanding the space-of-all-molecules.

The targets for this study were benzene, cubane, purine, diazepam, morphine and cholesterol. All targets contain rings and thus generate a search space with local optima. The fitness function can not only find similar molecules, which is useful in drug design, but can also lead evolution to the exact molecule used as the target. This can prove that the algorithm can find particular molecules. In addition, the number of generations to find the target provides a crude quantitative measure of performance.

Digital Logic

For digital logic circuit evolution, we attempted to evolve correct 15-step delay, parity, and one-bit add serial circuits. By "serial circuit" we mean that only one bit is input and output at each time step. The fitness of a circuit was the percentage of wrong output bits generated when processing 100 random input bits. Thus, a score of zero indicated a perfect circuit and a score of one a totally incorrect circuit. The circuits were simulated by assuming that every device (vertex) and wire (edge) required unit processing time . While this is not particularly realistic, it is easy and quick to implement and is sufficient to exercise the directed graph crossover operator. No attempt was made to generate optimal circuits, a task of greater interest to the digital hardware community.

Some initial runs ran out of memory when the circuits became extremely large. To reward parsimony, a fitness penalty of one percent was assessed against a circuit for each additional edge or vertex the circuit grew above a certain size. The penalty-free size was chosen to be well above that necessary to create the circuit.

Implementation

All computational experiments were run using the Condor cycle-scavaging batch system [Litzkow, et al. 1988] managing approximately 150 SGI workstations at NASA's NAS supercomputer center [Globus, et al. 2000]. Condor watches a "pool" of desktop workstations. When a workstation appears to be unused, Condor will match a waiting job with the workstation. The workstation will run the job until mouse or keyboard activity is detected, or the non-Condor CPU load exceeds a certain value. The job is then removed from the workstation. The job will execute later on (usually) another workstation. Jobs typically save their state periodically or when they are informed that they must leave a workstation. Saving state is called checkpointing. Checkpointing allows jobs to restart near where they left off. JavaGenes checkpoints periodically, usually at 30 minute intervals. With Condor, workstations that would otherwise be idle can perform useful computation.

JavaGenes is implemented in Java. Java was chosen because its syntax is similar to C++, many useful libraries are available (the graph layout software [Tunkelang 1998] and some statistics code were contributed by others), garbage collection vastly simplifies memory management, and Java?s array bounds checking and other bug-limiting features seem to substantially reduce debugging time and produce more robust code than C++, Fortran or C. A run-time penalty is paid for these advantages, but a generation of 500 individuals typically takes less than a minute to compute. The only significant performance problem has been with Java serialization. JavaGenes used serialization to checkpoint. Serialization is the process of flattening a data structure into an array of bytes, a form that can be saved to disk. Serialization is a standard part of the Java language. Although standard Java serialization worked well for small data structures, when the data structures grew large, serialization could take hours. Standard Java serialization was replaced with custom checkpoint/restart code, reducing checkpoint/restart time to around 10 seconds in most cases with a maximum around five minutes. See [Globus, et al. 2000] for a more detailed investigation of this problem.

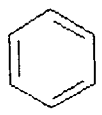
Long serialization times created a serious problem. In the initial implementation, JavaGenes started a new random number generator after restart. Because of this, evolution did not follow the same path taken after the last checkpoint. The genetic algorithm was controlled by different random numbers and therefore searched a different part of the space-of-all-graphs. Because the number of generations to find a specific target was used as the performance measure, re-starting the search repeatedly after a checkpoint made the algorithm appear more efficient than it was. Consider the case where a job ran for 100 generations up to a checkpoint and 20 more generations before being killed. When the job was restarted, a different set of populations would be constructed and one of the new generations, say the 10th, might find the target. Therefore, it would appear as if the target was found in 110 generations, but actually 130 populations were generated. Thus, when checkpoint times were long compared to the time available for execution on a single workstation, jobs might repeatedly search the region around the last checkpoint and stop if a solution were found, reporting an inaccurately small number of generations to completion. It was therefore necessary, at job restart time, to restart the random number generator with the same seed and then execute the random number generator until the sequence was where it left off. This should have insured that jobs followed the same evolutionary path regardless of checkpoint history. Unfortunately, although this fix worked properly when a job was restarted on the same machine, different results were observed in normal execution on the Condor pool, where jobs frequently move between machines. The changing evolutionary path was presumably caused by differences in the Java libraries on different versions of the SGI operating system. This may change the order of certain lists in JavaGenes and cause the same random number to pick a different item from a list. While this problem may make the results reported below somewhat optimistic for the larger molecules and circuits, we believe that the effect is fairly minor now that checkpointing is fast. There should be little effect on the smaller molecules because most jobs completed before the first checkpoint. Because checkpointing was fast, computations would usually proceed to the next checkpoint before being stopped and restarted. Only ~6% of all generations were repeated once the checkpointing performance problem was corrected.

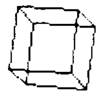
There was an additional problem during circuit evolution that was also caused by checkpointing. Each time a job restarted, a different set of random binary inputs was chosen to evaluate candidate circuits. This could have the effect of changing the fitness of a given circuit, although usually the change would be minor if the input sequence was long. Nonetheless, individuals generated after a restart could find themselves with a lower or higher fitness with respect to an older individual than if the restart had not taken place. We do not believe that this had any significant effect on the results.

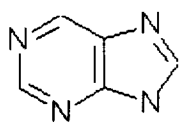
Test environment

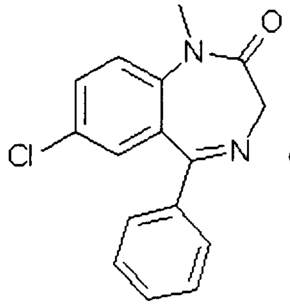
Molecules

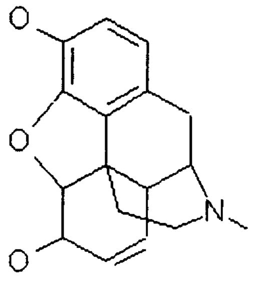
To see if JavaGenes could find molecules of interest, we tried to find the following targets:

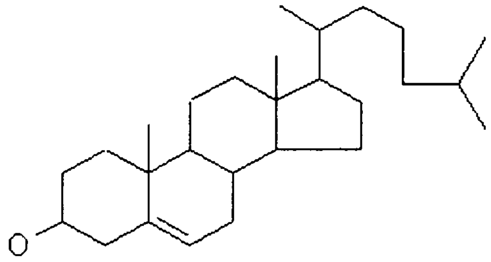
1.  benzene (C_6H_6) a simple ring molecule.

2.  cubane (C_8H_8) a cage molecule.

3.  purine ($C_5H_4N_4$) fused rings and heteroatoms.

4.  diazepam ($C_{16}H_{13}ClN_2O$) used in [Carhart, et al. 1985].

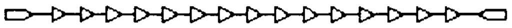
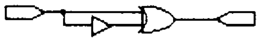
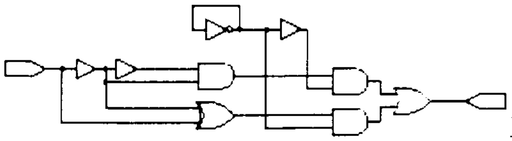
5.  morphine ($C_{17}H_{19}NO_3$) Dr. Wipke's group has worked on morphine analog design for many years.

6.  cholesterol ($C_{27}H_{46}O$) a non-drug molecule.

Stereochemistry and hydrogens are left out of the molecular diagrams since JavaGenes does not believe in them.

Digital Logic

We attempted to evolve thee different serial circuits:

-  15-unit delay
-  parity
-  1-bit add. Note that this circuit wouldn't work in hardware, but because our simulator assumes unit propagation delays for every device and wire, the *simulated* circuit will perform properly.

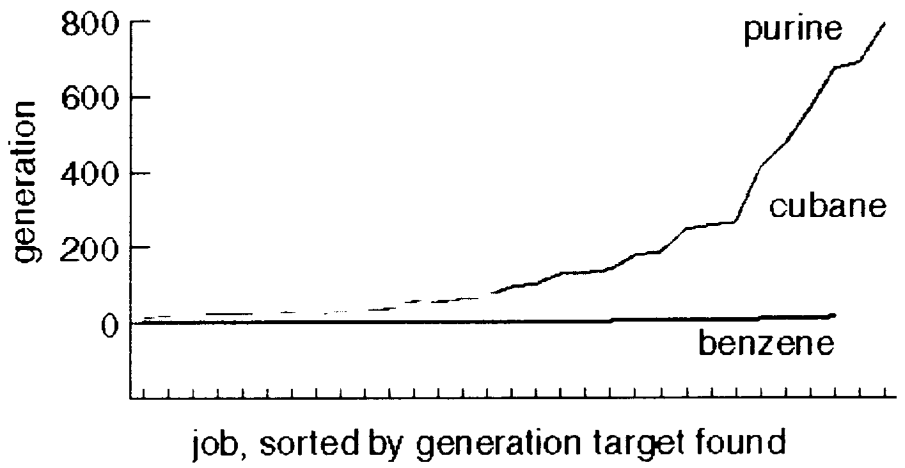
Results

Since the algorithm is stochastic, 31 jobs (each job is one execution of JavaGenes) were conducted for each experiment run. The number of generations and population size were varied. Once the target was found, jobs stopped. Jobs also stopped after a fixed, maximum number of generations. We use the number of generations to find a perfect individual as a performance measure. Combined with the population size, this provides a quantitative measure of performance. although the precise value of the numbers should not be taken too seriously. Even with 31 jobs per run. variation between runs with identical input parameters (other than the random number seed) was observed (see the comparison of purine and diazepam runs below). We display the results with line graphs where each data point is the generation one job of a run found a perfect individual; in other words. fitness was 0. The horizontal axis is the jobs, sorted by the generation a perfect individual was found. The verticle axis is the generation. Thus, all curves will be monotonically increasing.

Finding Small Molecules

First we compare JavaGenes performance finding three small molecules using a population size of 25 and a maximum of 1,000 generations:

Finding Benzene, Cubane, and Purine

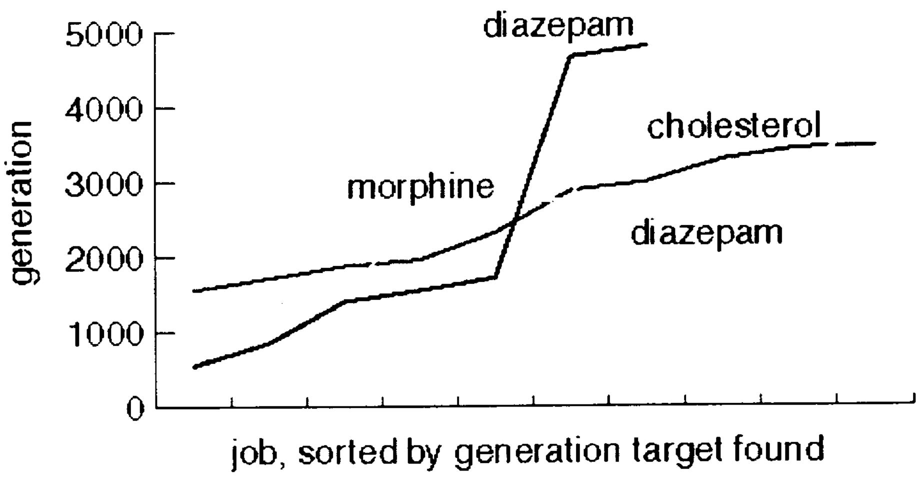


Note that two benzene jobs did not find the target at all, even though benzene was usually found in just a few generations. The jobs that could not find benzene lost all of the cycles in the population in the first generation created by crossover. When a population consists entirely of non-cyclic molecules, the crossover operator can not generate cycles.

Finding Larger Molecules

Now we compare JavaGenes performance finding three larger molecules using a population of 500 and a maximum of 5,000 generations. Note that there are two separate runs looking for diazepam.

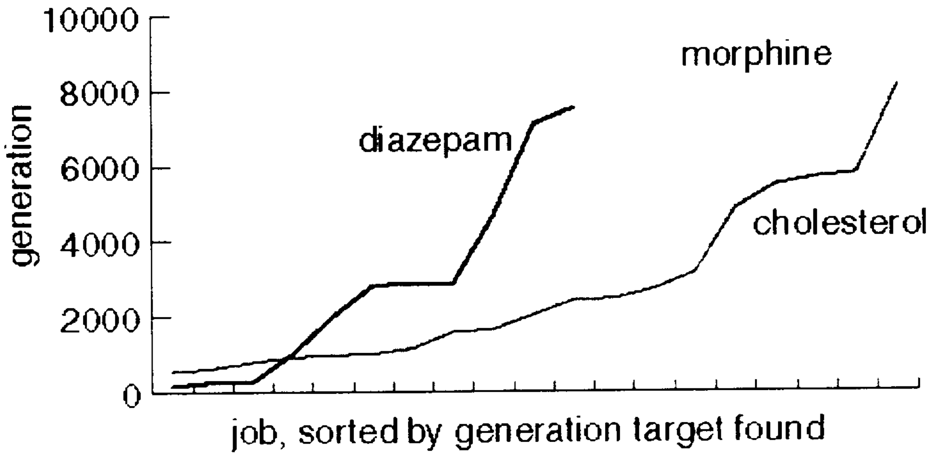
Finding Diazepam, Morphine, and Cholesterol



Although each run consisted of 31 jobs, only 7 jobs found morphine and 10 jobs found cholesterol. The two diazepam runs found the target 7 and 10 times respectively. Note that JavaGenes performance is much poorer on these larger molecules. Presumably, this is because the size of the space-of-all-molecules explodes combinatorially as molecule size increases. We also noticed that some populations lost all of the rare elements in the target. For example, diazepam has only one chlorine and one oxygen atom. Several of the diazepam jobs lost all of one these elements from the population and were forever doomed. Interestingly, jobs that lost all oxygen and/or chlorine from the population did so within about 400 generations. Using vertex mutation should avoid this situation, but this paper is concerned with the properties of the crossover operator.

Now we compare JavaGenes performance finding the same three molecules using a population of 1,000 and a maximum of 10,000 generations.

Finding Larger Molecules with a Larger Population

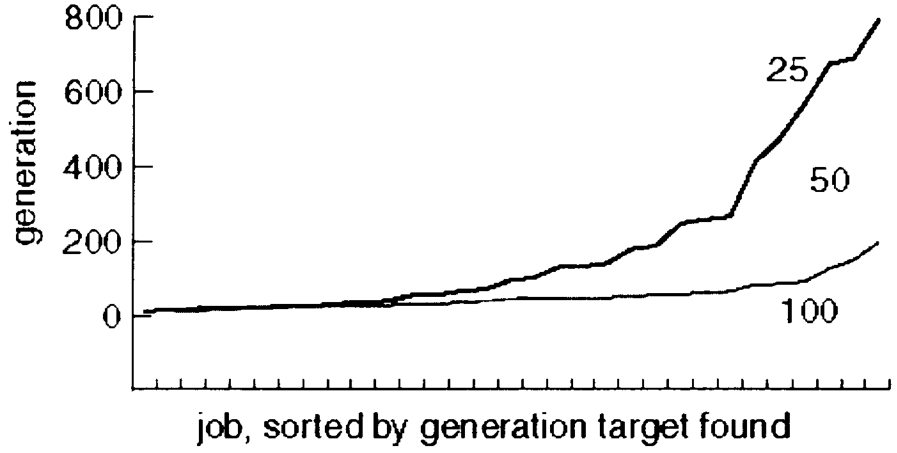


Performance is improved with a larger population and more generations, but 20 jobs still failed to find diazepam, 17 couldn't find morphine, and 12 failed to find cholesterol.

Effects of Population Size

We now examine the effects of population size by showing the results of searching for purine with a population size of 25, 50, and 100:

Finding Purine With Different Population Sizes

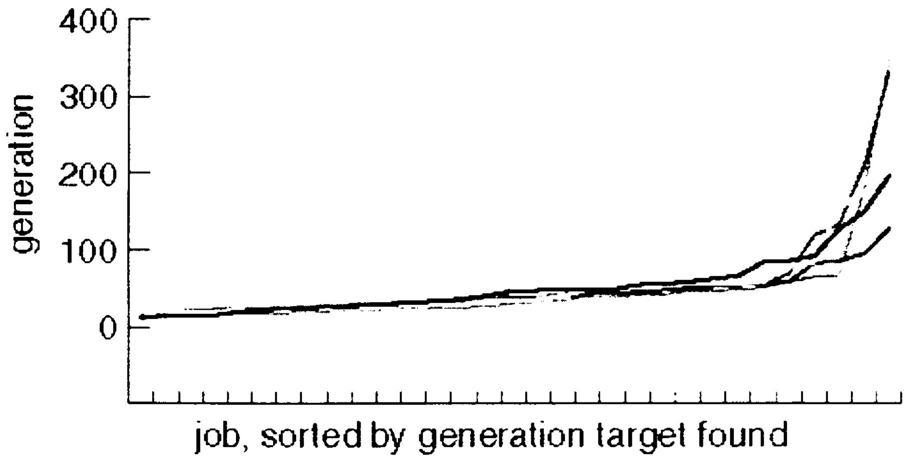


As expected, increasing the population size improves performace. Interestingly, with a population size of 25 and 50, the last job to find the target molecule took about the same number of generations. This is probably a random event, suitable for publication in the *Journal of Irreproducable Results.*

Variability Between Runs

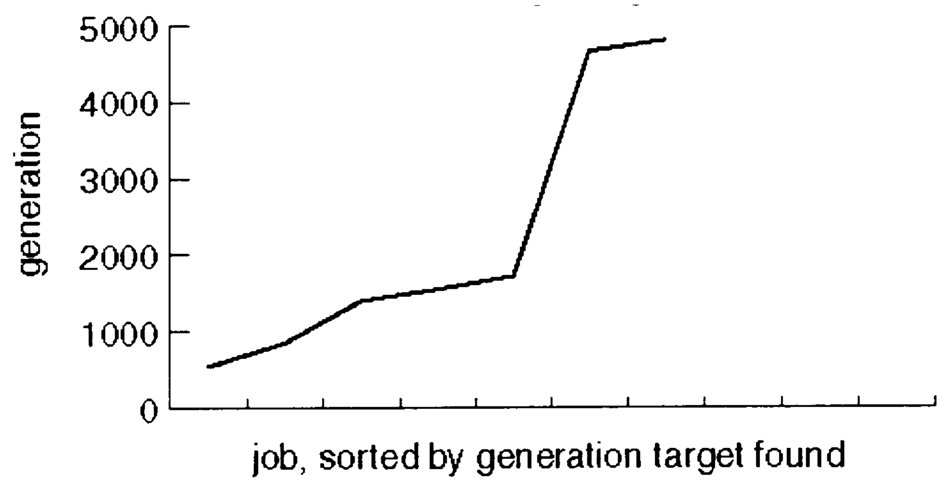
Since each job uses a different random number seed, results are somewhat different. To illustrate this variability, we made five runs looking for purine using identical parameters (except the random number seed). Population size was 100.

Identical Runs Finding Purine



The variability between runs is quite small for this molcular target. Most of the difference appears in the last two or three jobs to find purine from each run. To see if variablity was different with a more difficult target, we compare two runs searching for diazepam using a population size of 500 and a maximum of 5,000 generations:

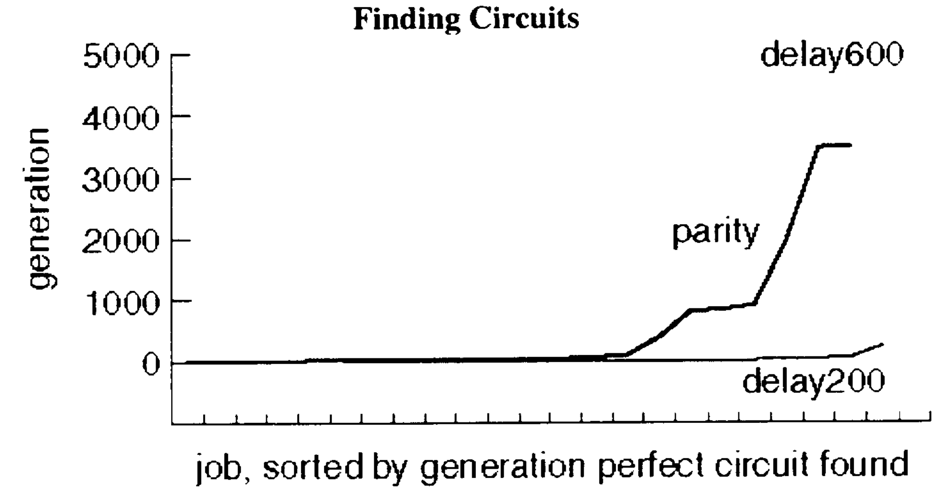
Identical Jobs Finding Diazepam



Note that although each run consisted of 31 jobs, only 7 and 10 jobs, respectively, were able to find diazepam in 5,000 generations. Nonetheless, although one run clearly out-performed the other, the results of the two runs are roughly comparable.

Finding Circuits

JavaGenes was able to successfully find small circuits that implement delay and parity functions. We compare results finding parity using a population of 600 and finding delay with populations of 200 and 600. In all cases, the maximum number of generations was 5,000:

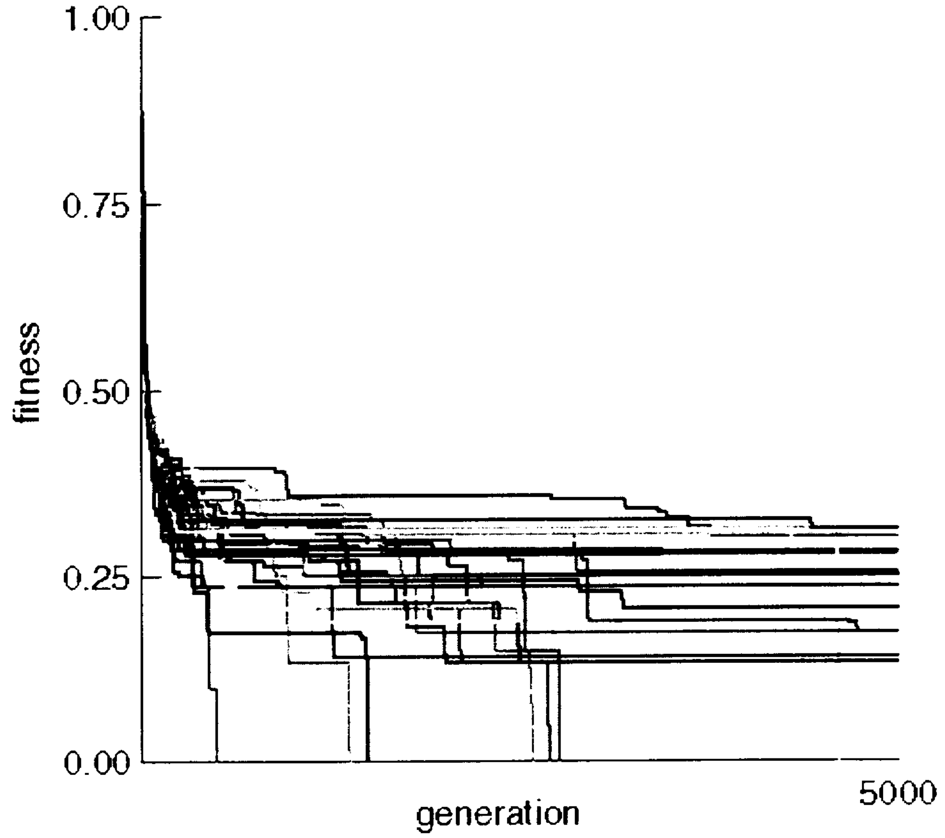


Note that only 22-24 (out of 31) jobs succeeded in finding a proper circuit in each run. Results are very similar for most of the successful jobs with substantial variability only for the worst performing jobs that succeeded. In spite of many attempts, we were never able to evolve a perfect 1-bit add circuit. The source of the problem is unclear.

Progress of a Run

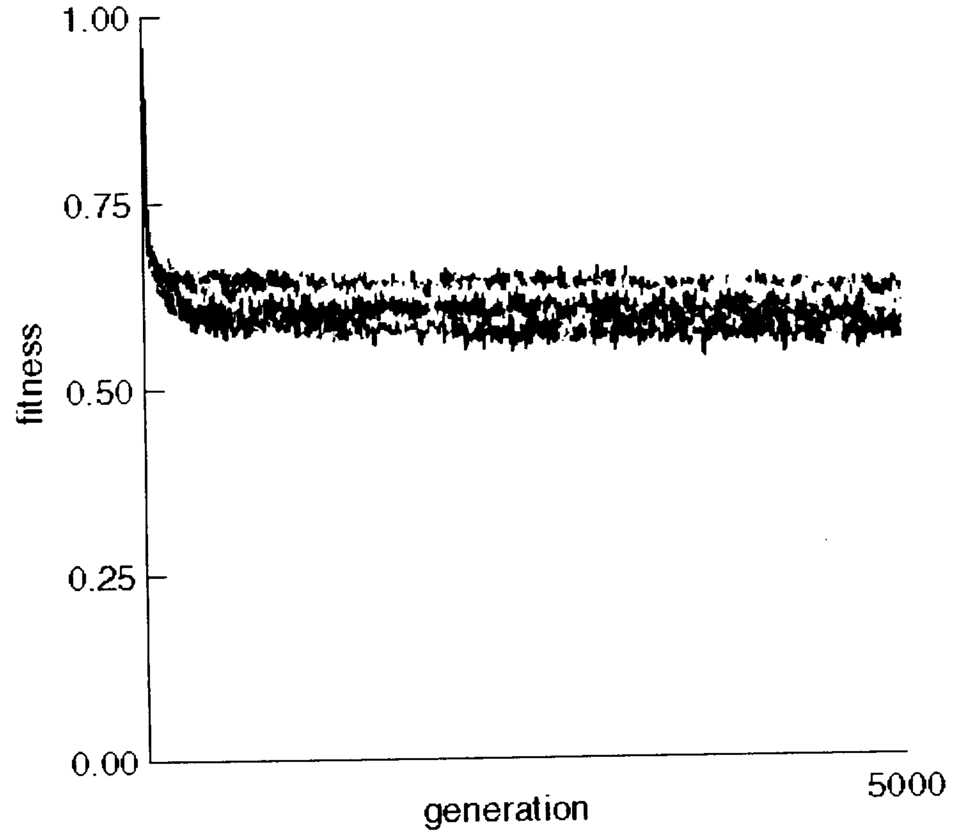
This figure shows the fitness of the best individual for each of 31 jobs searching for morphine. Population size was 500 and the maximum number of generations allowed was 5,000. Each data point was 10 generations apart. In other words, fitness data were collected from the population every 10 generations.

31 Jobs Searching for Morphine



The initial random populations all had a best individual fitness near 0.9, but quickly inproved to around 0.25. Most jobs then leveled off with long periods of no improvement, occasionally punctuated by sudden bursts of increasing fitness. Notice that morphine was often found even when the previous best fitness was quite poor, as evidenced by the long verticle lines ending at the x-axis.. This indicates that fitness improved very rapidly over the course of a few tens of generations.

We now examine the mean fitness of each job in the same run:



The mean fitness of each job dropped rapidly from about 0.96 to somewhere near 0.6 and stayed there with only very minor improvement. This may be caused by the extremely destructive nature of the crossover operator, which can be expected to generate many very unfit children from fit parents. Since every generated child was placed back into the fixed-size population. there was no guarantee that the individual replaced had lower fitness than the child. It might be interesting to develop a procedure that rarely replaces individuals if the child has worse fitness. One might expect the average fitness to continue to improve as evolution proceeds.

Effect of the "Coin Flip" Step

As mentioned before, [Globus, et al. 1999] provided results from the crossover algorithm, but with a bug that effectively eliminated the "flip coin" step in fragment combination. The results discussed above used code with additional modifications beyond the bug fix, primarily in the way that the initial population was generated. Table 1 compares performance before and after the bug fix, with no other code modifications. Input parameters were identical except for the number of jobs per run and the random number seeds for each job. The numbers in parentheses refer to results from [Globus, et. al 1999].

Table 1: Finding Small Molecules With and (Without) "Coin Flip" Step

31 runs for each molecule (20 with bug)	Population size	Median generations to find target	Minimum generations to find target	Number of runs that failed to find target	Maximum generations
Benzene	100 (200)	3 (39.5)	0 (2)	0 (8)	10 (1000)
Cubane	100	20 (46.5)	4 (13)	0 (0)	140(NR)
Purine	100	38 (245)	6 (19)	0 (4)	269(1000)

NR = not recorded. Because of job-to-job variability, and because many runs did not complete, median, rather than mean, generations to find the target is used, a procedure suggested by [Claerbout and Muir 1973].

Diazepam, morphine, and cholesterol were never found more than once each in [Globus, et al. 1999]. The difference in results before and after the bug fix show the importance of the "flip coin" step.

Comparison with Random Search

The great fear of search algorithm research is that, after months or years of effort, one's cherished algorithm will do no better than random search. To see if our crossover operator was better than random search, we searched for purine under three conditions: crossover alone, generating random molecules using the same algorithm as for the initial population (random search), and a 50-50 mix of crossover and random search. Twenty-one runs of 1000 generations on a population of 200 were conducted in each case. The [Globus, et al. 1999] algorithm was used. The fixed algorithm should do even better.

Comparison With Random Search

case	number of runs that found purine	median generations to find purine
random search	0	N/A
crossover alone	21	37
50-50 mix of crossover and random search	21	48

Clearly the crossover operator is better than random search.

JavaGenes can clearly find molecules and simple circuits. The algorithm consistently finds molecules but has great difficulty with circuits. Even the simple delay and parity circuits were not always found.

Summary

Algorithms and software to evolve graphs using genetic algorithm techniques were developed and applied to pharmaceutical drug and digital logic circuit design. Results suggest that the software can indeed discover a variety of molecules and very simple circuits. Significant additional work will be required to determine if applying genetic algorithms using a graph representation to molecular design is beneficial, but our results are encouraging. Unfortunately, our results suggest that digital circuit design may be extremely difficult using JavaGenes or similar software. Even the simple delay and parity circuits were difficult to evolve. Our results do, however, demonstrate that genetic algorithms can be applied to directed graphs.

Chemists have known for over a century that graphs are the most natural representation for molecules, just as logic designers use graphs to represent circuits. Furthermore, the space-of-all-graphs is not well understood or characterized. Therefore, it is reasonable to presume that searching for graphs using genetic algorithms will be profitable in a number of domains. We hope that our crossover operator will make a contribution.

Acknowledgments

Many thanks to Daniel Tunkelang, formerly of Carnegie Mellon, for providing his Jiggle code [Tunkelang 1998]. Jiggle arranges arbitrary graphs for easy viewing. Thanks to NWP Associates, Inc. for providing their Student T-Test code. Thanks to Rich McClellan, University of California at Santa Cruz, for providing the mol file reading and atomic element code. Thanks to Gail Felchle for much of the graphics art work. Thanks to ??? for reviewing this paper. This work was funded by NASA Ames contract NAS 2-14303.

References

[Baeck, et al. 1997] Thomas Baeck, Ulrich Hammel, and Hans-Paul Schwefel, "Evolutionary Computation: Comments on the History and Current State," *IEEE Transactions on Evolutionary Computation*, volume 1, number 1, pages 3-17, April 1997.

[Carhart, et al. 1985] Raymond Carhart, Dennis H. Smith, and R. Venkataraghavan, "Atom Pairs as Molecular Features in Structure-Activity Studies: Definition and Application," *Journal of Chemical Information and Computer Science*, volume 23, pages 64-73.

[Cheeseman, et al. 1991] Peter Cheeseman, Bob Kanefsky, William M. Taylor, "Where the Really Hard Problems Are," *Proceedings of the 12th International Conference on Artificial Intelligence*, Darling Harbor, Sydney, Australia, 24-30 August 1991.

[Claerbout and Muir 1973] J. F. Claerbout and F. Muir, ''Robust Modeling with Erratic Data,'' *Geophysics*, volume 38, pages 826-844.

[Corey and Wipke 1969] E. J. Corey and W. Todd Wipke, "Computer-Assisted Design of Complex Organic Syntheses," *Science*, volume 166, pages 178-192, 10 October 1969.

[De Jong 1990] K. A. De Jong "Introduction to the Second Special Issue on Genetic Algorithms." *Machine Learning*, 5 (4), 1990.

[Forrest and Mitchell 1993] Stephanie Forrest and Melanie Mitchell. "What Makes a Problem Hard for Genetic Algorithm? Some Anomalous Results in the Explanation," *Machine Learning*, volume 13, pages 285-319.

[Globus, et al. 1999] Al Globus, John Lawton, and Todd Wipke, "Automatic Molecular Design Using Evolutionary Techniques," *Nanotechnology*, volume 10, number 3, September 1999, pages 290-299.

[Globus, et al 2000] Al Globus, Eric Langhirt, Miron Livny, Ravishankar Ramamurthy, Marvin Solomon, Steve Traugott, "JavaGenes and Condor: Cycle-Scavenging Genetic Algorithms," submitted to Java Grande 2000.

[Holland 1975] John H. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press.

[Kinnear 1994] Kenneth E. Kinnear, Jr., "A Perspective on the Work in this Book," *Advances in Genetic Programming*, edited by Kenneth E. Kinnear, Jr., MIT Press, Cambridge, Massachusetts, pages 3-20, 1994.

[Koza 1992] John R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, Massachusetts, 1992.

[Koza, et al. 1997] John R. Koza, Forrest H. Bennett III, David Andre, Martin A. Keane and Frank Dunlap, "Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming," *IEEE Transactions on Evolutionary Computation*, volume 1, number 2, pages 109-128, July 1997.

[Koza, et al. 1999] John R. Koza, Forrest H. Bennett, Martin A. Keane, *Genetic Programming III : Darwinian Invention and Problem Solving,* Morgan Kaufmann Publishers, ISBN: 1558605436.

[Litzkow, et al. 1988] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - a Hunter of Idle Workstations,'' *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, June1988. See http://www.cs.wisc.edu/condor/.

[Lohn and Colombano 1998] Jason D. Lohn and Silvano P. Colombano, "Automated Analog Circuit Synthesis Using a Linear Representation,'' *Second International Conference on Evolvable Systems: From Biology to Hardware*, Springer-Verlag, 23-25 September 1998.

[Nachbar 1998] Robert B. Nachbar, "Molecular evolution: a Hierarchical Representation for Chemical Topology and its Automated Manipulation," *Proceedings of the Third Annual Genetic Programming Conference*, University of Wisconsin, Madison, Wisconsin, 22-25 July 1998, pages 246-253.

[Quarles, et al. 1994] T. Quarles, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE 3 Version 3F5 User's Manual*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, CA, March 1994.

[Teller 1998] Astro Teller, "Algorithm Evolution with Internal Reinforcement for Signal Understanding," Computer Science PhD Thesis, Carnegie Mellon University, Publication Number: CMU-CS-98-132.

[Tunkelang 1998] Daniel Tunkelang, A Numerical Optimization Approach to Graph Drawing, Dissertation, Carnegie Mellon University, School of Computer Science, December 1998.

[Weininger 1995] David Weininger, "Method and Apparatus for Designing Molecules with Desired Properties by Evolving Successive Populations," U.S. patent US5434796, Daylight Chemical Information Systems, Inc.